An Analytical Model of the Working-Set Sizes in Decision-Support Systems

Magnus Karlsson
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0969
U.S.A.
magnus_karlsson@hpl.hp.com

Fredrik Dahlgren
Ericsson Mobile Communications AB
Mobile Phones and Terminals
SE–221 83 Lund, Sweden
fredrik.dahlgren@ecs.ericsson.se

Per Stenström
Department of Computer Engineering
Chalmers University
SE–412 96 Göteborg, Sweden
pers@ce.chalmers.se

ABSTRACT

This paper presents an analytical model to study how working sets scale with database size and other applications parameters in decision-support systems (DSS). The model uses application parameters, that are measured on down-scaled database executions, to predict cache miss ratios for executions of large databases.

By applying the model to two database engines and typical DSS queries we find that, even for large databases, the most performance-critical working set is small and is caused by the instructions and private data that are required to access a single tuple. Consequently, its size is not affected by the database size. Surprisingly, database data may also exhibit temporal locality but the size of its working set critically depends on the structure of the query, the method of scanning, and the size and the content of the database.

1. INTRODUCTION

Decision-support systems (DSS) are one of the key drivers of the high-performance server market [17] taking the form of uniprocessor or parallel servers ranging from shared-nothing to shared-everything systems [5]. While I/O latency is an important performance obstacle in these systems, the increasing speed gap between processor and memory technology has also made good cache performance critical. Because DSS queries often access large (Gigabyte-sized) databases, it is presently unclear how well they exploit cache-memory resources and what controls the potential temporal locality inherent in typical workloads.

Temporal locality is intimately associated with reuse of chunks of instructions or data. Reuse of such chunks often results in distinct workings sets and a typical workload consists of many such working sets. Based on multiprocessor executions of scientific and technical workloads, Rothberg *et al.* found that the performance-critical working sets form a *hierarchy* [15]. In fact, they noticed that the most performance-critical working set easily fits in moderately-sized caches and it grows very slowly with the problem size thanks to the compute-intensive nature of the workloads. It is presently an open issue whether these observations extend to data-intensive workloads such as DSS. A fundamental understanding of the temporal locality inherent in DSS can potentially help designers of database engines and servers to find optimizations that result in substantially higher performance.

Cache-performance issues for data-intensive applications, e.g. DSS, have been studied previously [2; 4; 6; 8; 9; 11; 19; 21]. While temporal-locality issues are addressed in [2; 8; 9; 11; 21], these studies unfortunately focused on a fixed-sized database and did not bring much insights into how the working sets are affected when the database is scaled up. Moreover, since the code and data that contribute to the workings sets were not identified, previous studies have not gained any fundamental insights into capacity limitations of caches for DSS. This paper addresses both these shortcomings by providing an in-depth study on the working set growth in DSS and an analytical model that predicts the size of the most performance-critical working sets for a wide range of query compositions, database sizes, and database engines.

Our approach to identify the working sets inherent in DSS executions is to study the reuse that takes place in two database engines whose source codes we have access to (PostgreSQL and MySQL) on a full-system simulation platform (SimICS [12]). In this process, we isolated the application parameters (the size of the database and structural properties of DSS queries) that affect the size of the performance-critical working sets. Because simulation is too slow to model how the size of the working sets grow with the scale of the databases, we have developed an analytical model of the miss-ratio components whose parameters are carefully chosen so that they can be measured on down-scaled executions. We have validated our model against query executions on detailed system simulation models for the two database engines driven by queries from TPC-D and have found that our analytical model accurately predicts the miss ratios.

The major contributions in this paper are severalfold: The first contribution is a conceptual model of DSS query processing in which application parameters that critically control temporal locality are identified. We also believe that our analytical modeling approach is novel in that it allows working set hierarchies for a wide variety of queries, database contents and sizes to be studied in an efficient way. The third contribution is the observations we have made using the modeling machinery. We found that even for large databases, the most performance-critical working set fits in moderately large caches and is caused by the instructions and private data that are required to access a single tuple. The size of this working set does not depend on the database size. Surprisingly, database data may also exhibit temporal locality but the size of its working set critically depends on the structure of the query, the method of scanning, and the size and content of the database.

In Section 2, we first introduce a conceptual model allowing us to reason about what affects temporal locality in DSS queries. Based on this, we present and validate the analytical model in Sections 3 and 4, respectively. We then use the model and present our experimental findings in Section 5. The paper is ended with a discussion of related work in Section 6 before we conclude in Section 7.

2. TEMPORAL LOCALITY IN DSS

Typically, the same chunks of instructions or data are accessed many times in an execution. The *footprint* of such a chunk is the set of addresses that are accessed. The *size* of this footprint is an important parameter because it tells us exactly how big a single-word block size, fully-associative cache must be to keep this footprint. If the same chunk is accessed again, we say that its footprint is *reused*, and forms a *working set* with a size as big as the footprint. The size of the cache should be sufficient to match the sizes of the most performance-critical working sets.

This section identifies the most important working sets in DSS query processing and the parameters that control their sizes. First, Section 2.1 presents the execution flow in typical DSS queries and the data structures involved. Section 2.2 then identifies the critical code and data components that are potentially reused which forms the working sets. Application parameters that control their sizes is the topic of Section 2.3.

2.1 Query Processing in DSS

In a DSS based on the relational database model database data is split into tables called *relations* which consist of a number of database entries, called *tuples* [7]. A request sent to the database system is called a *query*. Each query sent to the DSS is converted into an optimized execution plan called a *query plan*, in which each node represents an operation and the arcs define the order in which the operations are executed. The operations of the query plan are then executed and a result is produced. A DSS query typically has a large number of operations that perform read-only operations on huge databases. Thus, the size and structure of the query plan has a great impact on the execution time.

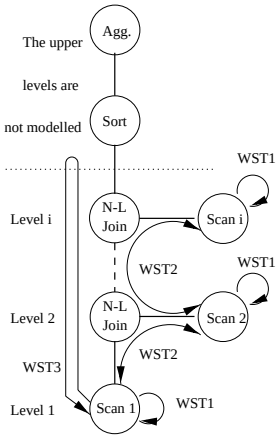
The operations in a query plan are typically to *scan* a relation for a matching tuple, to *join* the outputs of matched tuples from two scan operations, or to do some post-processing (aggregate, group and sort) of the output from scan and join operations. Because the part of the query plan that does the scanning and joining may dominate the execution time for complex queries on huge databases, we will only focus on these operations in the rest of the paper.

Scan operations can be done in mainly two ways: *index scan* or *sequential scan*. In a sequential scan, all tuples in a relation are accessed and its execution time is proportional to the size of the relation. Index scan, on the other hand, reduces the number of tuples to access by using indexes that point to the relevant part of the relation. A join node takes two relations of matched tuples as input and produces an output relation that contains all the tuples of the two relations that match the search criterion. There are three types of joins: (1) *nested-loop join* which for every matching tuple in one relation searches for tuples to join in the second relation; (2) *merge join* which merges the matching tuples of two sorted relations; and (3) *hash join* which constructs a hash table for the bigger relation and scans the other relation to perform the join.

We now focus on the data structures involved in the execution of a query. These data structures can be classified into six query data structures: *instructions, private data, meta data, index data, tuple locks,* and *database data.* Private data consist of private variables and relations. When shared tuples need to be modified or reordered only locally, as in sort nodes, shared tuples are copied into a private relation. Meta data consists of locks other than tuple locks, hash tables, buffer descriptors and other control structures. These locks are mostly spin-locks that are used to hinder more than one processor to access certain data, such as the tuple locks. The hash tables are used to find buffers and locks, while the buffer descriptors are the control structures for the shared buffer of tuples. Index data contains index structures to locate tuples. Tuple locks are the locks used to lock individual tuples in a relation. Consequently, a larger number of such locks are accessed than locks in the meta-data category. Finally, database data consist of all the relations of the database.

The left diagram of Figure 1 shows an example query plan. It consists of i scan nodes using the sequential scan method, $i-1$ nested-loop join nodes, and a sort and an aggregate node. To the right in Figure 1, the pseudo code of a nested-loop join is shown. Note that even though the discussion that follows focuses on nested-loop joins, we will later extend the discussion to handle hash joins, merge joins and index scans. This is explained at the end of Section 2.3.

The execution starts with the topmost join being called with `level` set to i. The relation is scanned until either a tuple matches the search criterion or there are no more tuples to access. When a match occurs, and if there are more levels to scan, nested-loop join is recursively called to scan another relation at the level below. On the other hand, if there are no more levels to scan, a join occurs which joins the matching tuples from all the higher levels and sends them on to the post-processing phase to be sorted for example. When no more tuples are found at a certain level, the rest of the tuples at the level above are scanned for the next matching tuple.




```
nested_loop_join(level)
begin
   start from the beginning of the relation;
   while there is still a tuple left to access in the relation
      get the next tuple;                          (S2)
      while the tuple does not match the search criterion and
                  we have not reached the end of the relation
         get the next tuple;                       (S1)
      endwhile

      if the tuple matches the search criterion
         if (level = 1)
            perform the join operation;            (S3)
            get the next tuple;                    (S3)
         else
            nested_loop_join(level-1);
         endif
      endif
   endwhile
end
```

Figure 1: A query plan and pseudo code for nested-loop joins.

2.2 Working-Set Transitions

A frequent operation in a typical query is the access of an individual tuple. Accessing a tuple is associated with a set of well-defined footprints of each of the query data structures. Because this operation is so frequent, the question is what parts of the individual footprints are reused between two consecutive tuple accesses. In order to analyze this, we must distinguish between all the different ways by which two consecutive tuples can be accessed. These are illustrated in the nested-loop join example in Figure 1, and are referred to as *working-set transitions*.

To start with, if the first tuple does not match, the next tuple in the same relation will be accessed. This working-set transition is denoted *WST1* and is depicted as transitions to statement S1 in the pseudo code. The second case is when two consecutive tuples are accessed in different scan nodes. This working-set transition is called *WST2* and is depicted as transitions to statement S2. It occurs when a new scan node is invoked either when there is a match on levels higher than 1 or when all tuples in the relation have been scanned. Finally, the third case is applicable only when a tuple in a level-1 node matches the search criterion. Then a join operation takes place before the next tuple is accessed. This working-set transition is called *WST3* and is depicted as a transition to either of the two statements denoted S3. In Table 1, we show what footprints of the query data structures are reused for each working-set transition by denoting unconditional reuse with 'U' and conditional reuse with 'C'.

WST1 is common in sequential scan operations. Because the same instructions and set of private variables are accessed across tuple accesses in WST1, their footprints are always reused. Moreover, the same locks in the meta data are always reused. By contrast, the footprint of database data, indexes, and tuple locks are only reused across consecutive invocations of the same scan node and not across tuple accesses during the same invocation. For WST2, instruction footprints are only reused if the two scan operations use the same scanning method. Private data are only reused if the same scan node has been invoked before. Because we have found that most of the locks in the meta data will be reused, meta data is marked as being reused. In contrast, the other data structures can only be reused if the same scan node is invoked again. Finally, WST3 involves the scan node at the lowest level. While a part of the instruction, private data, and meta data footprint is associated with the join, we disregard its reuse characteristics because the size of its footprint turns out to be negligible in comparison with the footprint caused by the scan node. Therefore, WST3 has similar reuse behavior as WST1.

Table 1: Reuse of the query data structures for the three working set transitions. 'U' and 'C' correspond to unconditional reuse and conditional reuse across consecutive invocations of the same scan node, respectively.

Data Structure	WST1	WST2	WST3
Instructions	U	C	U
Private Data	U	C	U
Meta Data	U	U	U
Index Data	C	C	C
Tuple Locks	C	C	C
Database Data	C	C	C

2.3 Working Sets in DSS queries

Let us now identify the working sets that show up for each data structure and for each working-set transition. We refer to a working set with its name, e.g. WS, and to its size with $|WS|$.

Because instructions, private data, and, meta data are reused for WST1 and WST3, they form a well-defined working set denoted MWS. $|MWS|$ is dictated by instructions, private data, and meta data needed to access a single tuple. Meta data accesses in WST2 are also satisfied by MWS. The instruction footprint differs between index scan and sequential scan. This means that MWS only suffices for instructions if WST2 involves two scan nodes of the same kind. By contrast, private data under WST2 form a working set of their own. Suppose that WST2 makes a transition to a node at level i. Since this node was visited last, all the scan nodes at levels j below ($j < i$) have been invoked. Each invocation causes a new set of private data with a unique footprint P_j whose size is $|P|$. Thus, the cache size must be at least $i \cdot |P|$ to host all these sets of private data. We call this working set *private data working set* and refer to it as PWS_i.

Table 2: Minimum cache size requirements to guarantee reuse for each data structure at level i.

Data Structure	WST1	WST2	WST3
Instructions	$\|MWS\|$	$\|MWS\|$	$\|MWS\|$
Private Data	$\|MWS\|$	$\|MWS\| + \|PWS_i\| + \|DWS_{i-1}\|$	$\|MWS\|$
Meta Data	$\|MWS\|$	$\|MWS\|$	$\|MWS\|$
Index Data	$\|MWS\| + \|DWS_i\|$	$\|MWS\| + \|PWS_i\| + \|DWS_i\|$	$\|MWS\| + \|JWS\| + \|DWS_i\|$
Tuple Locks	$\|MWS\| + \|DWS_i\|$	$\|MWS\| + \|PWS_i\| + \|DWS_i\|$	$\|MWS\| + \|JWS\| + \|DWS_i\|$
Database Data	$\|MWS\| + \|DWS_i\|$	$\|MWS\| + \|PWS_i\| + \|DWS_i\|$	$\|MWS\| + \|JWS\| + \|DWS_i\|$

Index data, tuple locks, and database data have the same reuse characteristics because a tuple access involves locating, locking, and then accessing the tuple. The working set associated with them, called *database working set* (DWS), is therefore the same. To understand what is contained in this working set, consider a query plan containing only sequential scan nodes. Owing to the recursive nature of the query, the set of tuples that has been accessed between two consecutive accesses to the same tuple at level i is essentially all relations at lower levels. Let the number of tuples accessed at level i be N_i, then $\sum_{j=1}^{i} N_j$ distinct tuples are contained in this working set.

It is interesting to note that queries containing sequential scans form distinct working sets with respect to index data, tuple locks, and database data. On the other hand, for index scan methods, we cannot be sure that the same N_i tuples are accessed at level i on two consecutive invocations. In order to model index-scan methods, we introduce a parameter called *database-reuse factor* (DB_RE_i), which is interpreted as the fraction of the relation that is accessed each time the scan node is invoked. Thus, given that N_i tuples are accessed at each invocation of a scan node at level i, the number of tuples in the relation is N_i/DB_RE_i. As a result, the working set contains $\sum_{j=1}^{i} N_j/DB\_RE_j$ tuples.

Let us now consider the minimum cache size requirements to fit the working sets involved in a tuple access at level i. For WST1, space must be enough to fit $|MWS| + |DWS_i|$. In addition to this, WST2 needs space for private data; its size is $|MWS| + |PWS_i| + |DWS_i|$. Finally, in WST3, joining data involves some additional footprints for the last three data structure categories. This footprint is reused and we refer to the working set as JWS. Consequently, WST3 requires a cache as big as $|MWS| + |JWS| + |DWS_i|$. We summarize the cache size requirements in Table 2.

Since a query execution involves transitions between different WSTs, we need to model how these transitions affect the average working set size during the whole execution. Our approach is to weight the size of the working set associated with each WST with its relative frequency. The relative frequencies are controlled by three application parameters: the number of tuples accessed in a relation at level i (N_i); the probability that an accessed tuple matches the search criterion at level i (H_i); and the depth of the query tree (QD). If N_i is high and H_i is low, as often is the case when sequential scanning is used, the frequency of WST1 will dominate the execution. On the other hand, if H_i is high, as when index scan is used, the relative frequency of WST2 will be high. In Table 3 the equations for the relative frequencies are tabulated. For example, the probability that a WST1 occurs at level 1 for a sequential scan is the probability that a match does not occur and there is another tuple to scan. Since there are $(N_1-1)(1-H_1)$ such tuples, this probability is $\frac{(N_1-1)(1-H_1)}{N_1}$.

Table 3: The probability that a certain working set transition will occur on level i.

Level (i)	WST 1	WST 2	WST 3
$i = 1$	$\frac{(N_1-1)(1-H_1)}{N_1}$	$\frac{1}{N_1}$	$\frac{H_1(N_1-1)}{N_1}$
$i > 1$	$\frac{(N_i-1)(1-H_i)}{N_i}$	$\frac{1}{N_i} + \frac{H_i(N_i-1)}{N_i}$	0

Let us finally extend the discussion to other join methods than nested-loop joins. If hash joins are used, the actual join and scan phase can be described by our model, but the construction of the hash tables is not modeled. If the construction of hash tables dominates the execution of the hash-join, our model will not accurately predict the cache miss ratio. A hash join will have $H = 1$, N will be equal to the number of matching tuples, and DB_RE_i will be set to its minimum value, which is N_i/S_i where S_i is the size of the relation at level i, because there is no reuse. Merge joins can be handled by changing the H_i and DB_RE_i parameters. H_i is now the probability that the next tuple will be merged from the other relation, and DB_RE_i will be set to N_i/S_i.

3. ANALYTICAL MODEL

This section presents the analytical model for the cache miss ratio as a function of the size of a fully-associative cache. We first present the overall approach in Section 3.1 before we present the model equations in Section 3.2.

3.1 Overview of the Model

Based on scientific applications, Rothberg *et al.* [15] found that all working sets in an execution typically form a hierarchy like the one sketched in Figure 2. This diagram shows the miss ratio versus the cache size assuming a fully-associative cache. While a distinct drop in the miss ratio rarely shows up in practice, it is useful for the continued discussion that such distinct "knees" really exist.

The lower bound on the miss ratio is defined by the cold miss ratio. As for the capacity miss ratio, the sudden drops conceptually mean that a working set suddenly fits into the cache. The combined effect of all working sets forms the working-set hierarchy. To understand where the distinct drops in the miss ratio occurs, we note that a tuple access at level i results in accesses to MWS and DWS_i. Therefore it becomes meaningful to consider the working set composed of the union of these working sets. Because $(MWS \cup DWS_i) \subset (MWS \cup DWS_j), i < j$ it follows that $| MWS \cup DWS_i | < |MWS \cup DWS_j |$. The capacity misses associated with tuple accesses at level i then form the plateau in the diagram of Figure 2 denoted DWSi with size $| MWS | + | DWS_i |$.

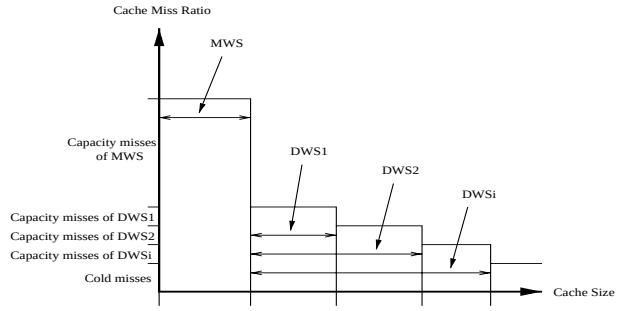


Figure 2: Example working-set hierarchy.

As we will see in the next section, the height of the plateau is the cold miss ratio plus the sum of the capacity miss ratio components of each DWS.

3.2 Model Equations

The model equations use three sets of parameters: *measured* parameters (footprint sizes and number of accesses at the tuple level); *application* parameters (description of the query composition as described in Section 2); and *system* parameters (e.g. cache size). Since the intuition is that the measured parameters are invariant to the size of the database and the structure of the query, they can be measured on a down-scaled run. Based on these parameters and the working set hierarhcy model described above, we derive equations for the number of cold misses and capacity misses to be discussed next.

3.2.1 Number of cold misses

The number of cold misses for data structure DS is trivially determined by dividing the entire footprint size for the query execution by the cache block size.

$$Cold\ Misses(DS) = \frac{footprint\ size\ of\ the\ data\ structure}{cache\ block\ size}$$

We measure the size of the footprints involved in a single tuple access for each data structure. These footprints are from both the database engine and the operating system. The size of the instruction footprint is measured. Recall from Section 2.3 that the size of the footprint of private data is $|P_0|$ plus $QD \cdot |P|$, where QD is the number of levels in the query. We measure $|P_0|$ and $|P|$ and calculate the size of the entire footprint based on the height of the query. The size of the footprints for database data, index data, and tuple locks is proportional to the number of tuples accessed in the query. By code inspection, we could determine the size of the footprint associated with each tuple access. It is then possible to calculate the size of the entire footprint by multiplying the footprint size at the tuple level with the number of tuples accessed which is $\sum_{i=1}^{QD} N_i/DB\_RE_i$. Finally, meta data also has one footprint whose size is calculated analogously and one constant part which, like private data, is determined through code inspection.

3.2.2 Number of capacity misses for MWS

To determine the number of capacity misses, we assume a fully associative cache with an LRU replacement policy. As will be shown in Section 4 this yields a good accuracy unless the associativity is too limited. One of the measured parameters in the model is $|MWS|$ which is assumed to keep the instruction footprint plus the space needed for private, index, tuple lock, meta data and database data to process a single tuple. As a result, when the size of the cache is $|MWS|$, it is expected to not cause any capacity misses for instructions, private data, and meta data for WST1 and WST3 according to Section 2.

Capacity misses decline dramatically in number when going from no cache to a cache with only a few block entries. Since we are primarily interested in caches larger than normal first-level caches, we base our model on measurements of the number of capacity misses per tuple access on a small cache ($C_0 = 16$ KByte). Such a cache is expected to exploit all the inherent spatial locality in a tuple access. As we increase the cache size, there is a higher probability that the requested instructions, private and meta data remained in the cache since the last tuple access. We assume a rectangular distribution of this probability. Thus, the number of capacity misses to MWS per tuple access, $CM_{MWS}(DS, C)$, for data structure DS as a function of the cache size C is

$$\begin{cases} 0 & ; C > |MWS| \\ CM_{MWS}(DS, C_0) \cdot (1 - \frac{C - C_0}{|MWS| - C_0}) & ; C_0 \le C \le |MWS| \end{cases}$$

where $CM_{MWS}(DS, C_0)$ is the measured number of capacity misses per tuple access for data structure DS on a system with a 16-KByte cache.

In order to determine the total number of capacity misses, we have to calculate the total number of tuple accesses. This can be calculated based on N_i (the number of scanned tuples at level i), H_i (the probability that a tuple matches the search criterion at level i), and QD (the number of levels). The scan node at the highest level is invoked once, the scan node at level i is invoked $\prod_{j=i+1}^{QD} N_j \cdot H_j$ times and accesses N_i tuples. Thus, the total number of tuples is

$$NumOfTuples = \sum_{i=1}^{QD} N_i \prod_{j=i+1}^{QD} N_j \cdot H_j$$

3.2.3 Number of capacity misses for DWS

According to Section 2, more cache space than $|MWS|$ is needed to also keep the working sets associated with index data, tuple locks, and database data. We consider the capacity misses caused by DWS_i in this section.

If the cache is not large enough to contain both MWS and DWS_i, there will be capacity misses at the invocation of the scan node. Looking at one working-set transition, the critical issue is whether the footprint of all accesses since the last time the same tuple was previously accessed will fit in the cache. If it does not, the accessed tuple has been replaced owing to the sequential nature in tuple accesses and its interaction with the LRU-policy. Therefore, the number of misses experienced by an access to index data, tuple locks, and database data, respectively, is the size of the entry accessed (E) divided by the block size. These system parameters are assumed to be known in the model. Thus, the number of capacity misses for each tuple access to data structure DS assuming WST1, denoted $CM_{DWS}(DS, i, WST1, C)$, is

$$\begin{cases} 0 & ; C \ge |MWS| + |DWS_i| \\ E/B & ; |MWS| \le C < |MWS| + |DWS_i| \\ \frac{E}{B} + CM_{MWS}(DS, C) & ; C_0 \le C \le |MWS| \end{cases}$$

To calculate the total number of capacity misses involving WST1 at level i, we have to determine the number of WST1s in the query which is the total number of tuples times the probability of a WST1 according to Table 3:

$$No\ of\ WST1\ at\ level\ i = \frac{(N_i - 1)(1 - H_i)}{N_i} \cdot N_i \prod_{j=i+1}^{QD} N_j \cdot H_j$$

To calculate the number of capacity misses caused by tuple accesses for WST2 and WST3 follows the same strategy with the following exceptions. First, the cache size intervals are changed according to Table 2 and to calculate the number of WST2 and WST3 we use their probabilities from Table 3. Finally, we have to sum the capacity misses from each WST category which forms the number of capacity misses caused by DWS_i in Figure 2. Finally, in order to calculate the total number of capacity misses, we have to sum all the misses across all data structures and levels.

3.2.4 Number of memory accesses

The total number of accesses is calculated as the total number of of tuple accesses (see above) times the mean number of accesses per WST ($Accesses_{WST}$).

$$Accesses = NumOfTuples \cdot Accesses_{WST}$$

The last parameter depends on the database engine and the operating system and has a measured value. While we have assumed that the number of accesses per WST is the same for all WST types, our validations have shown this to be a reasonable estimation. The cache miss ratio can now be calculated as the number of misses divided by the number of accesses.

4. VALIDATION

In this section we will validate our analytical model by focusing on the validity of the model in three critical areas: (1) across different database engines and queries; (2) across different database sizes; and (3) across caches with different degrees of associativity. We first introduce the validation set-up in Section 4.1 and then present the validation results in Section 4.2.

4.1 Simulated System

The simulated system consists of four parts: a database workload, a database engine, an operating system, and a simulated hardware platform. The database workload consists of three queries from TPC-D [3]: Q3, Q6, and Q10. These queries were chosen because they represent two of the join methods (hash join and nested-loop join) and both the scan methods available in the two database engines we studied. Note that TPC-D is not used for benchmarking in this study. It is used only to provide relevant queries and database contents to experiment with. The two database engines are PostgreSQL [22] v 6.1.1 and MySQL [18] v 3.22. The operating system is Linux v 2.0.35. All software is compiled with gcc -O2.

We simulate a uniprocessor system implementing the Sun4m model using the SimICS platform [12]. This platform implements a functional model including a SPARC V8 instruction set simulator, memory semantics, and device drivers that are sufficient to support a Sun4m port of the Linux operating system. In order to measure the cache miss ratio, we use a fully-associative cache with a line size of 64 bytes, and with a zero-cycle memory access time. This system model is connected to the SimICS platform. The traces produced from the platform are fed into the system model on the fly.

Table 4: Application parameters for Q3, Q6 and Q10 as a function of cache size. S is the database size and T the average size of the part of a tuple that is accessed.

Parameter	Q3	Q6	Q10
QD	3	1	4
N_i	$\frac{S}{T \cdot 45}$, $\frac{S}{T \cdot 1000}$, $\frac{S}{T \cdot 50000}$	$\frac{S}{T \cdot 1.45}$	$\frac{S}{T \cdot 45}$, $\frac{S}{T \cdot 200}$, $\frac{S}{T \cdot 2000}$, 1
H_i	0.2, 1, 1	0.1	0.15, 1, 1, 1
DB_RE_i	1, $\frac{T \cdot 50000}{S}$, 1	1	1, $\frac{T \cdot 2000}{S}$, 1, 1
T	290	234	280

We start collecting statistics to compute the measured parameters when the database engine starts to access the first tuple and stop right after the last tuple is accessed.

4.2 Validation Results

The first validation is based on PostgreSQL. The topmost diagrams of Figure 3 show the measured and predicted miss ratios versus the cache size for Q3, Q6 and Q10 assuming a 20-MByte database. Due to the prohibitively slow simulations of PostgreSQL, we could not afford to run larger databases. However, the validations on MySQL, reported hereafter, are performed on a 200-MByte database. The model parameters have all been measured on a 5-MByte database. Apart from showing the total miss ratio, we also break it down in per data-structure miss-ratios, where each miss-ratio component is calculated as the number of misses from that component divided by the number of accesses to the component. The index and the tuple lock data structures have been lumped into database data because their impact is negligible. The solid lines are the measured miss ratios whereas the dashed lines are the miss ratios predicted by the model. The application parameters for the queries are shown in Table 4.

Q3 consists of one sequential scan, two index scans and two nested-loop joins. In Q6 a large relation is sequentially scanned once from one end to the other. Q10, on the other hand, consists of two index scans, two sequential scans, two hash-joins and a nested-loop join. As can be seen from the figure, the model accurately predicts the measured cache miss ratios. The only noticeable discrepancy happens for meta data. This discrepancy is due to the fact that our model represents meta data as one collective data structure instead of many smaller distinct structures as mentioned in Section 2.1. However, the effect of this simplification is small on the overall cache miss ratio.

To see if our analytical model accurately predicts the miss ratio for another database engine, we have performed the validation on the commercial database system MySQL [18]. We ran the same queries as for PostgreSQL but with a 200-MByte database. Again, all the model parameters have been measured on a 5-MByte database. The bottommost diagrams in Figure 3 show the cache miss ratio of Q3, Q6 and Q10 on MySQL. Note that MySQL generates a query plan

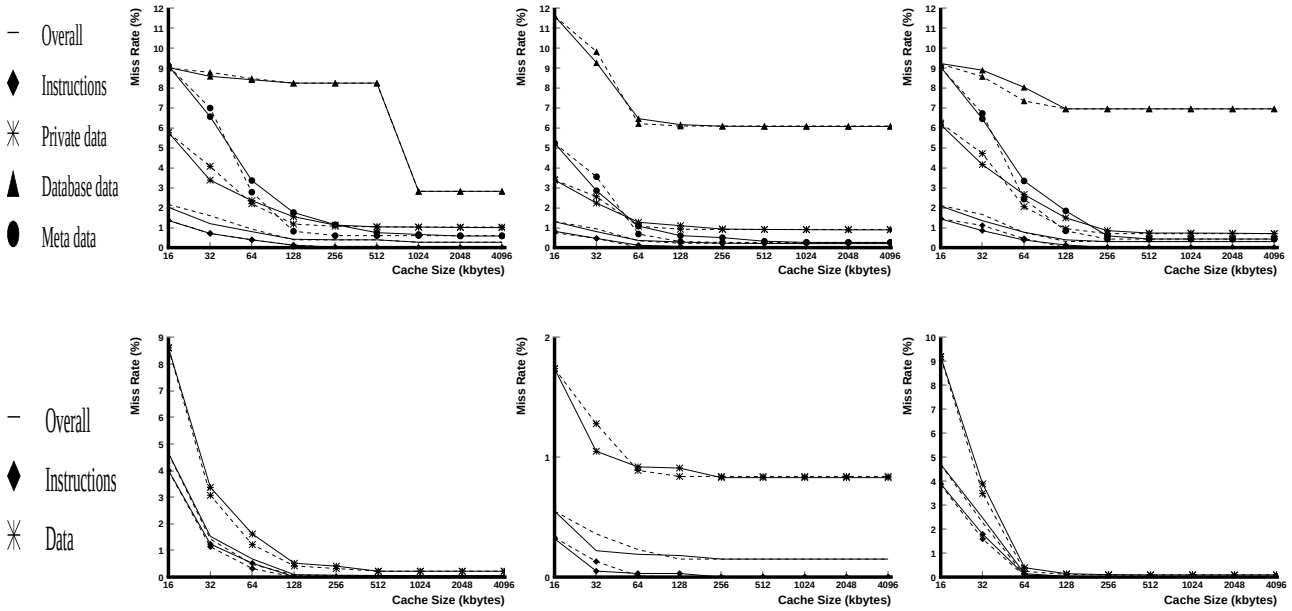


Figure 3: The cache miss ratios for Q3 (left), Q6 (middle), and Q10 (right). The top diagrams show results for a 20-MByte database run on PostgreSQL and the bottom diagrams show results for a 200-MByte database run on MySQL. The solid lines are simulated results and the dashed ones are modeled.

for Q3 that only contains index scans, in contrast to PostgreSQL that has one sequential scan. Also for this database engine our analytical model predicts the cache miss ratios accurately.

To investigate how well our model can predict the miss ratios of caches with limited associativity, we also ran our simulations with such caches. The results of these simulations are shown in Figure 4. First, despite the fact that we do not model cache conflicts, the model manages to fairly accurately predict the shape of MWS for all caches except for the direct-mapped. However, for cache sizes smaller than MWS, the conflict miss ratio appears to be quite big even for caches with limited associativity as clearly evidenced for Q6. In fact, the 2:1 cache rule of thumb seems to apply and dictates that at least a twice as big a cache is needed to host the working set for MWS than predicted by our model. Nevertheless, the model is still useful to understand what application parameters affect the size of the working sets which we will demonstrate in the next section.

5. MODEL EXPERIMENTS

In the previous section, we validated our model against detailed simulations for fairly small database sizes. In this section we assume larger database sizes, and study how the different working sets are affected by the behavior of the application. We start in Section 5.1 by showing the cache miss ratio of Q3 executing on a 10-Gigabyte database for various cache sizes. The results show that even on this big database the most performance-critical working set, MWS, of a DSS query is relatively small. Then, in Section 5.2, we examine how the query composition affects the sizes and cache miss ratios of the different working sets. One important characteristic of the query that strongly affects the execution is whether indexing is used in the scan operations. Therefore, Section 5.3 focuses on the difference in cache miss characteristics of sequential scan versus index scan. Finally, Section 5.4 generalizes the findings to other queries in TPC-D on uniprocessors and to parallel implementations of those queries.

5.1 The Most Critical Working Set

Figure 5 shows the miss ratios of Q3 on a 10-Gigabyte database for different cache sizes. The application parameters of Q3 are shown in Table 4. The diagram depicts the total cache miss ratio, where each data structure's individual contribution to the overall cache miss ratio is shown. The miss ratios for index data and tuple locks are not shown because their impact on the overall cache miss ratio is small.

The first observation is that MWS – the most critical working set – is small even for a 10-Gigabyte database. When the cache size is increased from 16 to 256 KByte, the miss ratio is reduced from about 2% to 0.3%. Instruction misses dominate the cache miss ratio for small cache sizes, but are virtually eliminated at a cache size of 128 KBytes. Private data and meta data also contribute to the cache miss ratio for small cache sizes, but at a cache size of 256 KBytes their impact is nullified. The second observation is that when the cache is large, the main contributor to the total cache miss ratio is database data. This suggests that miss reduction or hiding techniques should target database data.

Results from half a GByte TPC-D runs on multiprocessors [2; 11] report larger cache miss ratios for these cache sizes. This is due to mainly three effects: First, the caches they used had limited associativity; second, we are only running one database server process; and finally, we have no coherence misses as we are running on a single processor system.

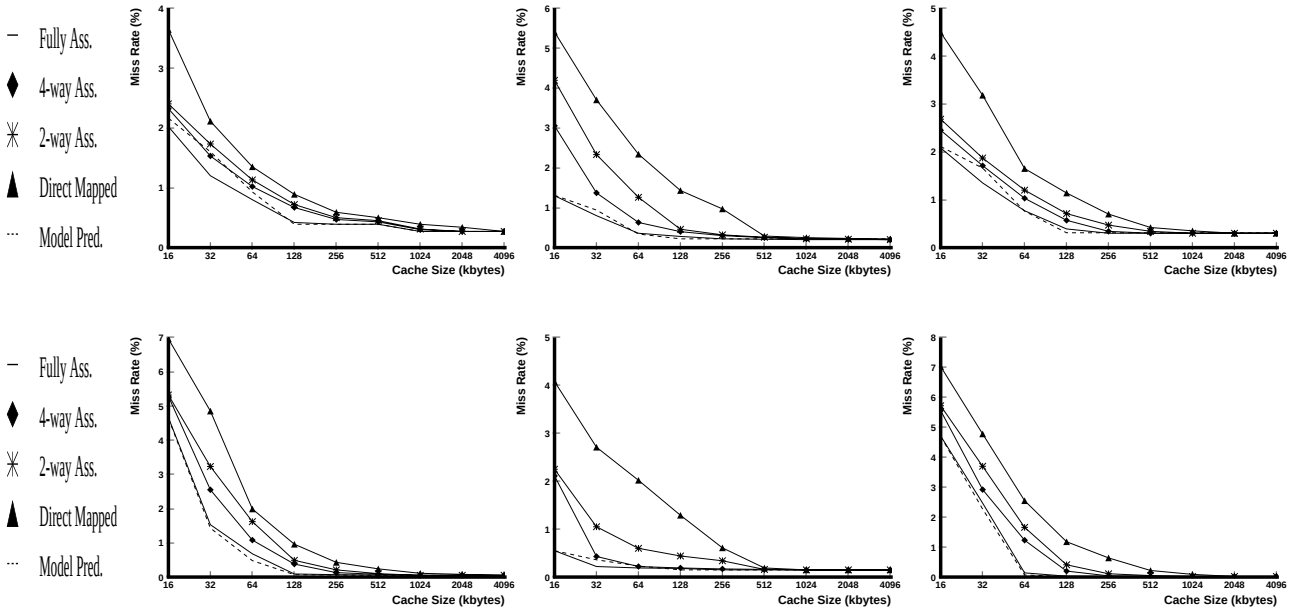


Figure 4: The correspondence between the model predictions and caches with limited associativity. The top and bottom diagrams show results for PostgreSQL (20 MB) and MySQL (200 MB), respectively. From left to right results are for Q3, Q6 and Q10.

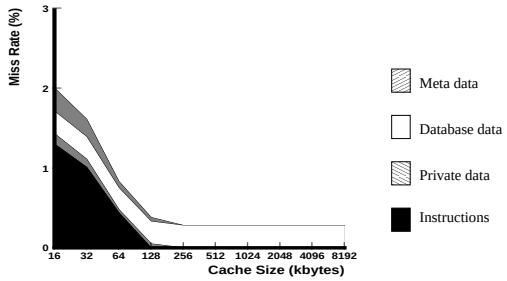


Figure 5: The cache miss ratio of Q3 for a 10-GByte database subdivided into four data structures according to their contribution to the number of cache misses.

5.2 Impact of Query Composition

In this section, we explore how the application parameters we identified in Section 2 affect the size of the working sets using the model from Section 3. We focus on the following four application parameters: N_i - the number of tuples scanned per invocation of scan node i, H_i - the probability that a tuple matches the search criterion in node i, QD - the depth of the query tree, and DB_RE_i - the fraction of the relation that is accessed each time the scan node is invoked. N_i and H_i are investigated because they are controlled by the database size, and the other two because their impact on the size of the working sets can be substantial. In this section, we focus on Q3 with the parameters according to Table 4 except the parameters that are varied in the experiments.

In the top left diagram of Figure 6, we show the cache miss ratio as a function of the cache size for different values of N_1. The set of curves at the bottom of this diagram corresponds to the total miss ratios whereas the set of curves at the top correspond to the miss ratios for database data. The N_1 values of the curves are 1000, 2000, 10000 and 100000, where a higher N_1 results in a higher miss ratio. Overall, we notice that this application parameter mainly affects the miss ratio of database data; the total miss ratio is only marginally affected. Database data miss ratio makes a distinct leap from 9% down to around 3%, and the cache size at which this happens depends on N_1 which determines the size of DWS1.

Q3 uses sequential scan at level 1, and N_1 corresponds to the size of the relation scanned at the lowest scan node. When the cache size is smaller than N_1, it cannot exploit temporal locality between different invocations of the scan node, and the miss ratio is determined by the spatial locality during one sequential scan. When the cache size is larger than the relation, the temporal locality in the relation scanned on level one will be utilized. In this case the scan node is invoked three times resulting in a decrease of the database data miss ratio from 9% to about 3%. For cache sizes smaller than DWS1, temporal locality could be utilized by alternating the order in which a relation is scanned, or using blocking strategies [10].

The next application parameter that can be affected by the database size is H_i. The top right diagram of Figure 6 shows from left to right the miss ratio of Q3 for the H_2 values of 1, 1/10, 3/100, and 1/100, for N_1 and N_2 values of 2000 and 100, respectively. Similar to the left diagram, it again shows both database data and total cache miss ratio, where a higher H_2 results in fewer misses. Since H_2 does not affect the size of the footprint of neither instructions, private nor database data, the sizes of MWS and DWS1 are unaffected. This can be seen in the diagram since the cache sizes at which the miss ratios show distinct reductions are the same for different values of H_2. However, H_2 affects the number of times the scan node on level 1 is invoked, and for cache

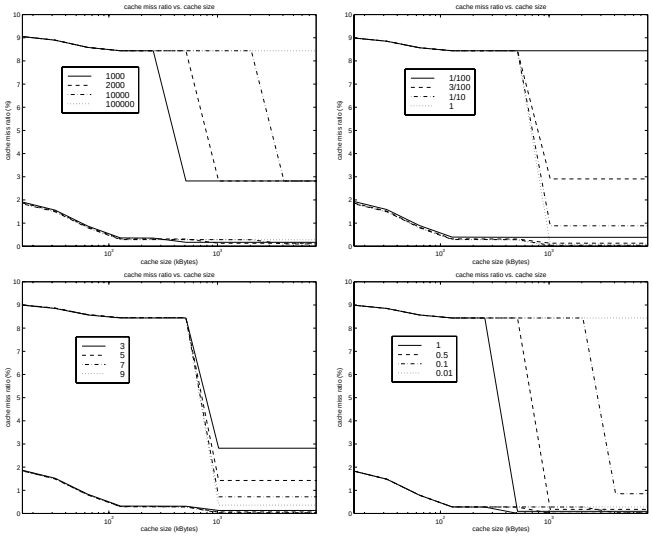


Figure 6: Top: The cache miss ratio of Q3 with N_1 values of 1000, 2000, 10000 and 100000 in the left diagram. The right one shows Q3 with H_2 values of 1, 1/10, 3/100 and 1/100. Bottom: Left diagram shows results for various query depths (3,5,7, and 9) and right diagram shows results for various DB_RE_1 values (1, 0.5, 0.1, and 0.01).

sizes larger than DWS1 the temporal locality can be utilized by the cache. Therefore, a higher H_2 value increases the hit ratio for cache sizes larger than DWS1.

Changing H_1 instead of H_2 would only affect the number of WST3 and consequently the amount of join operations performed. This will have only minor effects on the miss ratio. Varying H_3 would alter the number of invocations of the scan nodes at level 1 and 2, in the same way as varying H_2 changes the number of invocations of the scan node at level 1. In general, H_i for $i > 1$ affects the cache miss ratio for database data for caches larger than DWS_j for $j < i$.

Another application parameter that can have a significant effect on the miss ratios and working sets is the query depth QD, i.e. the number of scan and join nodes of the query. The bottom left diagram of Figure 6 shows the cache miss ratio for database data and the overall miss ratio for query depths 3, 5, 7 and 9. The query extends Q3 with added query nodes that have an N_i of $S/(T \cdot 2000)$, where S is the total size of the database and T the size of each tuple, an H_i of 1 and a minimum DB_RE_i. A higher query depth has a lower miss ratio in the figure.

The results show that QD, just like H_i, impacts on the miss ratios but not on the size of the working sets. The reason is that increasing the query depth often increases the number of times a scan node is invoked, and the temporal locality is thus increased which can be utilized by large enough caches. In the figure, database data only show one distinct "knee" on the miss ratio curve despite the multiple levels of the query. The reason for this is that N_1 is much larger than N_i for $i > 1$, which makes the plateaus caused by the other DWSs only marginally larger than DWS1.

The last application parameter that we focus on is the database reuse factor (DB_RE_i), i.e. the fraction of the relation that is accessed each time the scan node is invoked. For sequential scan the database reuse factor is always 1. Each time a scan node is invoked the whole relation is scanned from one end to the other. For index scans on the other hand, the database reuse factor depends on the database contents.

The bottom right diagram of Figure 6 shows Q3 with four different values of DB_RE_1; 1, 0.5, 0.1 and 0.01. N_1 is 1000 and H_2 is 1. The diagram only shows the cache miss ratio of database data and the overall ratio. The results show that higher DB_RE_1 for a fixed N_1 results in a smaller DWS1. Lowering the database reuse will increase the number of times a scan node has to be invoked in order for any database data to be reused. The trends are related to those shown in the left diagram of Figure 6 where N_1 was varied. The reason for this was discussed in Section 2; the quotient N_i/DB_RE_i determines the cache space needed to utilize the temporal locality of the database data of the scan node at level i.

In summary, the experiments presented in this section revealed the following. First, the size of MWS is neither affected by N_i nor H_i. This suggests that it is possible to run huge databases without getting high cache miss ratios for fairly reasonable sized caches. Second, while N_i and DB_RE_i affect the size of the working sets caused by database data, H_i and QD affect the magnitude of the miss ratios. This gives an intuition into how different queries affect the cache miss ratio.

5.3 Sequential Scan versus Index Scan

The cache miss behavior of sequential scan and index scan can be quite different. Sequential scan has a database reuse of 1, an often low probability of finding a matching tuple H_i, and a large N_i. For index scan the database reuse is often low, H_i is high, and N_i is usually small. Figure 7 shows the overall cache miss ratio for Q3 on a 50-MByte database, where the left and right diagrams show the results from using sequential and index scan methods in all scan nodes, respectively.

First of all sequential scan has a smaller $|MWS|$ than index scan due to the fact that sequential scan is a much less complex operation which requires less instructions, private data and meta data. Second, index scan has a higher cache miss ratio for small cache sizes due to a larger footprint of instructions, private data and meta data, and a higher ratio of cold misses versus accesses. Between 128-KByte and 1024-KByte caches, sequential scan has a higher cache miss ratio. The reason for this is a low utilization of the temporal locality of database data for cache sizes smaller than DWS1. For caches larger than DWS1, the cache miss ratio of sequential scan becomes lower than for index scan because of more temporal locality from the large reuse of data at the scan node on the lowest level. It has to be said that even though index scan has a higher cache miss ratio for large cache sizes, it executes a lot faster in most cases owing to fewer tuple accesses.

5.4 Generalizations: Queries and Systems

We have done the same experiments for Q6 and Q10 as for Q3. In this section, we will only mention the major findings.

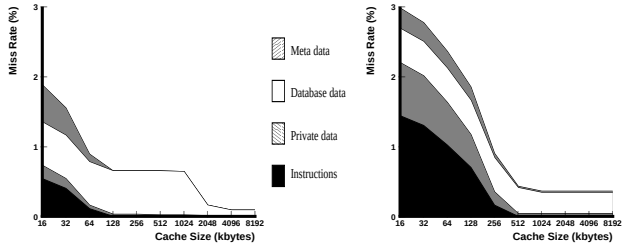


Figure 7: The total cache miss ratio of Q3 with sequential scan nodes to the left and with index scan nodes to the right.

Looking at the miss ratios for a 10-GByte database, the results are similar to those from Q3 in terms of the size of MWS, and the major differences are the actual miss ratios. Q6 contains only two query plan nodes, with one scan and one aggregate node. This has two consequences: the MWS is smaller and the miss ratios are lower than for Q3 and Q10 for cache sizes smaller than MWS. Overall, however, all the trends we have seen in the previous sections for Q3 also hold for Q6 and Q10.

The queries of TPC-D all have a query depth under nine. This fact alone implies that the size of MWS of all the TPC-D queries is expected to be small. Even if all the queries where dominated by WST2, the extra amount of private data would still be small.

Our analytical model can also be used to reason about parallel operations within a query. The most common parallel operations in the scan and join phase are *parallel scan, parallel join* and *parallel subtasking* [5]. In parallel scan, the data is subdivided into smaller relations and each of these subrelations are scanned at a different processor and the partial results are then merged. A parallel scan can be modeled by dividing N_i with the number of processors. However, if the merge phase is dominant this will be less accurate. Parallel join can be modeled in the same way. Therefore, it is possible to use the working set analysis framework to reason about cache size requirements for multiprocessors. We would expect that MWS is small also in such environments. Parallel subtasking will have the effect of cutting the depth of query plans.

6. RELATED WORK

Barroso *et al.* [2] and Lo *et al.* [11] studied both memory system issues for commercial DSS and commercial OLTP-system. They used a single database size and concluded that OLTP exercises the memory system more than DSS. Trancoso *et al.* [21] studied PostgreSQL running DSS queries for various L1 and L2 cache sizes, again using a single database size. They concluded that there is spatial locality in DSS queries but very little temporal locality. While all these studies have added to the insights of interactions between data-intensive workloads and memory systems, none of them have investigated how the temporal locality of a database application is affected by the size of the database. While [21] tied their observations to the application level, they unfortunately did their experiments on a small database.

The notion of working set hierarchies was introduced in [15]. However, the authors only examined scientific applications and concluded that the five application classes they studied have small working sets that virtually don't grow with input data size. This study has investigated whether these observations extend to DSS applications.

We also think that our analysis framework contributes to the literature on cache modeling. Thiébaut and Stone [20] examined the transient behavior of caches for two processes executing in a round-robin fashion. They did not account for reuse between processes and limit their discussion to two processes. In [13], the framework is extended to account for cache lines that are dead, i.e. has no reuse. However, they model the application as a fractal random walk. The analytical models presented in [1; 14] predict the cache miss ratio of a program. Some of the model parameters are measured from address traces. While both the models are very accurate, changing one application parameter of the DSS would change the address trace and thus several model parameters in an intricate way. Sorin *et al.* derived in [16] an analytical model for the number of instructions retired per cycle of an ILP processor. If the cache size is changed, they have to rerun their simulator to estimate some of the application parameters again. Our modeling framework does not have these shortcomings.

7. CONCLUSION

The key question that was addressed in this paper is what application parameters affect the sizes of the performance critical working sets for decision-support systems, especially when the size of the database is scaled up. In order to answer this question, we have investigated what components in query processing that potentially can exhibit good temporal locality and what application parameters that control the sizes of the working sets of these components.

By analyzing the PostgreSQL and MySQL database systems driven by queries from TPC-D, we developed a qualitative model that identifies the components that potentially are reused and what application parameters that control the size of their working sets. The components we have identified to cause well-defined working sets are the instructions and private data needed to access a single tuple and the database data. While the size of the former is independent of the database size, the latter depends on many application parameters such as the structure of the query, the method of scanning (index versus sequential), and the size and the content of the database.

In order to quantitatively study which working sets dominate in the TPC-D benchmark, we developed a novel analytical modeling approach that uses measurements from simulations as parameters allowing us to vary other parameters that critically control the working sets. We confirmed that even for large databases, the most performance-critical working set does not grow with database size and is caused by the instructions and private data that is required to access a single tuple. Surprisingly, database data may also exhibit temporal locality but the size of its working set critically depends on the structure of the query, the method of scanning, and the size and the content of the database.

Acknowledgments

We are indebted to Margaret Martonosi of Princeton for valuable feedback on an earlier manuscript. Other colleagues that have contributed with feedback are Erik Hagersten, Michael Koster, and Anders Landin of Sun Microsystems Inc. This research has been supported by grants and equipment from Sun Microsystems, the Swedish Strategic Research Foundation (SSF) under contract number 311-97-99, and the Swedish Council for Planning and Coordination of Research (FRN) under contract number 96238.

8. REFERENCES

[1] A. Agarwal, M. Horowits, and J. Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 3–14, June 1998.

[3] T. P. P. Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.1*, December 1995.

[4] Z. Cventanovic and D. Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 60–71, April 1994.

[5] D. Dewitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. In L. Hongjun, O. Beng-Chin, and T. Kian-Lee, editors, *Query Processing in Parallel Relational Database Systems*, pages 4–17. IEEE Computer Society Press, 1994.

[6] R. Eickemeyer, R. Johnson, S. Kunkel, M. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 203–212, May 1996.

[7] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 2 edition, 1995.

[8] A. Grizzaffi Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, October 1996.

[9] K. Keeton, D. Patterson, Y. Q. He, R. Raphael, and W. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 15–26, June 1998.

[10] M. S. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimization of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[11] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 39–50, June 1998.

[12] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of USENIX'98*, pages 119–130, June 1998.

[13] A. Mendelson, D. Thiébaut, and D. K. Pradhan. Modeling Live and Dead Lines in Cache Memory Systems. *IEEE Transactions on Computers*, 42(1):1–14, January 1993.

[14] R. W. Quong. Expected I-Cache Miss Rates via the Gap Model. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 372–383, April 1994.

[15] E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 14–25, May 1993.

[16] D. Sorin, P. S. Vijay, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic Evaluation of Shared-Memory Systems with ILP Processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 380–391, June 1998.

[17] P. Stenström, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal. Trends in Shared Memory Multiprocessing. *IEEE Computer*, 30(12):44–50, December 1997.

[18] TcX AB, Detron HB and Monty Program KB. *MySQL v3.22 Reference Manual*, September 1998.

[19] S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 228–238, May 1990.

[20] D. Thiébaut and H. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[21] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the 3rd International Conference on High Performance Computing*, pages 250–260, February 1997.

[22] A. Yu and J. Chen. The POSTGRES95 User Manual. In *Computer Science Division, Department of EECS*, University of California at Berkley, July 1995.